CPSC 330
Applied Machine Learning

Lecture 5: Preprocessing and `sklearn` pipelines

UBC 2022-23

Instructor: Mathias Lécuyer

Imports

In [1]:
```
import sys
import time

import matplotlib.pyplot as plt

%matplotlib inline
import numpy as np
import pandas as pd
from IPython.display import HTML

sys.path.append("../code/.")

import mglearn
from IPython.display import display
from plotting_functions import *

# Classifiers and regressors
from sklearn.dummy import DummyClassifier, DummyRegressor

# Preprocessing and pipeline
from sklearn.impute import SimpleImputer

# train test split and cross validation
from sklearn.model_selection import cross_val_score, cross_validate, tr
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    MinMaxScaler,
    OneHotEncoder,
    OrdinalEncoder,
    StandardScaler,
)
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from utils import *

pd.set_option("display.max_colwidth", 200)
```

Announcements

- Homework 3 is out (Due date: Feb. 1st). Please start early.!
- Homework 1 grades and solutions are out. Please do not share them with anyone or do not post them anywhere.

Learning outcomes

From this lecture, you will be able to

- explain the motivation for preprocessing in supervised machine learning;
- identify when to implement feature transformations such as imputation, scaling, and one-hot encoding in a machine learning model development pipeline;
- use `sklearn` transformers for applying feature transformations on your dataset;
- discuss the golden rule (a.k.a. don't look at the test set until the end) in the context of feature transformations;
- use `sklearn.pipeline.Pipeline` and `sklearn.pipeline.make_pipeline` to build a preliminary machine learning pipeline.

Motivation and big picture [video (https://youtu.be/xx9HlmzORRk)]

- So far we have seen
  - Three ML models (decision trees, k-NNs, SVMs with RBF kernel)
  - ML fundamentals (train-validation-test split, cross-validation, the fundamental tradeoff, the golden rule)
- Are we ready to do machine learning on real-world datasets?
  - Very often real-world datasets need preprocessing before we use them to build ML models.

Example: k-nearest neighbours on the Spotify dataset

- In lab1 you used `DecisionTreeClassifier` to predict whether the user would like a particular song or not.
- Can we use a k-NN classifier for this task?
- Intuition: To predict whether the user likes a particular song or not (query point)
  - find the songs that are closest to the query point
  - let them vote on the target
  - take the majority vote as the target for the query point

In [2]:
```
spotify_df = pd.read_csv("../data/spotify.csv", index_col=0)
train_df, test_df = train_test_split(spotify_df, test_size=0.20, random
X_train, y_train = (
    train_df.drop(columns=["song_title", "artist", "target"]),
    train_df["target"],
)
X_test, y_test = (
    test_df.drop(columns=["song_title", "artist", "target"]),
    test_df["target"],
)
```

In [3]:
```
dummy = DummyClassifier(strategy="most_frequent")
scores = cross_validate(dummy, X_train, y_train, return_train_score=Tru
print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
pd.DataFrame(scores)
```

```
Mean validation score 0.508
```

Out[3]:

	fit_time	score_time	test_score	train_score
0	0.004357	0.001195	0.507740	0.507752
1	0.000990	0.000347	0.507740	0.507752
2	0.000742	0.000344	0.507740	0.507752
3	0.000674	0.000306	0.506211	0.508133
4	0.000642	0.000300	0.509317	0.507359

In [4]:
```
knn = KNeighborsClassifier()
scores = cross_validate(knn, X_train, y_train, return_train_score=True)
print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
pd.DataFrame(scores)
```

```
Mean validation score 0.546
```

Out[4]:

	fit_time	score_time	test_score	train_score
0	0.013952	0.018358	0.563467	0.717829
1	0.002404	0.009834	0.535604	0.721705
2	0.002058	0.008475	0.529412	0.708527
3	0.002010	0.008687	0.537267	0.721921
4	0.001937	0.008455	0.562112	0.711077

In [5]:
```
two_songs = X_train.sample(2, random_state=42)
two_songs
```

Out[5]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mo
842	0.229000	0.494	147893	0.666	0.000057	9	0.0469	-9.743	
654	0.000289	0.771	227143	0.949	0.602000	8	0.5950	-4.712	

In [6]:
```
euclidean_distances(two_songs)
```

Out[6]:
```
array([[    0.        , 79250.00543825],
       [79250.00543825,     0.        ]])
```

Let's consider only two features: `duration_ms` and `tempo` .

In [7]:
```python
two_songs_subset = two_songs[["duration_ms", "tempo"]]
two_songs_subset
```

Out[7]:

	duration_ms	tempo
842	147893	140.832
654	227143	111.959

In [8]:
```python
euclidean_distances(two_songs_subset)
```

Out[8]:
```
array([[    0.        , 79250.00525962],
       [79250.00525962,     0.        ]])
```

Do you see any problem?

- The distance is completely dominated by the the features with larger values
- The features with smaller values are being ignored.
- Does it matter?
  - Yes! Scale is based on how data was collected.
  - Features on a smaller scale can be highly informative and there is no good reason to ignore them.
  - We want our model to be robust and not sensitive to the scale.
- Was this a problem for decision trees?

Scaling using `scikit-learn` 's [`StandardScaler` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScal](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)

- We'll use `scikit-learn` 's [`StandardScaler` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html), which is a `transformer` .
- Only focus on the syntax for now. We'll talk about scaling in a bit.

In [9]:
```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()  # create feature trasformer object
scaler.fit(X_train)  # fitting the transformer on the train split
X_train_scaled = scaler.transform(X_train)  # transforming the train sp
X_test_scaled = scaler.transform(X_test)  # transforming the test split
pd.DataFrame(X_train_scaled, columns=X_train.columns).head()
```

Out[9]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudn
0	-0.697633	-0.194548	-0.398940	-0.318116	-0.492359	1.275623	-0.737898	0.395
1	-0.276291	0.295726	-0.374443	-0.795552	0.598355	-1.487342	-0.438792	-0.052
2	-0.599540	1.110806	-0.376205	-0.946819	-0.492917	0.446734	-0.399607	-0.879
3	-0.307150	1.809445	-0.654016	-1.722063	-0.492168	0.170437	-0.763368	-1.460
4	-0.634642	0.491835	-0.131344	1.057468	2.723273	0.170437	-0.458384	-0.175

`fit` and `transform` paradigm for transformers

- `sklearn` uses `fit` and `transform` paradigms for feature transformations.
- We `fit` the transformer **on the train split** and then transform the train split as well as the test split.
- We apply the same transformations on the test split.

`sklearn` API summary: estimators

Suppose `model` is a classification or regression model.

```
model.fit(X_train, y_train)
X_train_predictions = model.predict(X_train)
X_test_predictions = model.predict(X_test)
```

`sklearn` API summary: transformers

Suppose `transformer` is a transformer used to change the input representation, for example, to tackle missing values or to scales numeric features.

```
transformer.fit(X_train, [y_train])
X_train_transformed = transformer.transform(X_train)
X_test_transformed = transformer.transform(X_test)
```

- You can pass `y_train` in `fit` but it's usually ignored. It allows you to pass it just to be consistent with usual usage of `sklearn` 's `fit` method.
- You can also carry out fitting and transforming in one call using `fit_transform`. But be mindful to use it only on the train split and **not** on the test split.

- Do you expect `DummyClassifier` results to change after scaling the data?
- Let's check whether scaling makes any difference for k-NNs.

In [10]:

```
knn_unscaled = KNeighborsClassifier()
knn_unscaled.fit(X_train, y_train)
print("Train score: %0.3f" % (knn_unscaled.score(X_train, y_train)))
print("Test score: %0.3f" % (knn_unscaled.score(X_test, y_test)))
```

```
Train score: 0.726
Test score: 0.552
```

In [11]:

```
knn_scaled = KNeighborsClassifier()
knn_scaled.fit(X_train_scaled, y_train)
print("Train score: %0.3f" % (knn_scaled.score(X_train_scaled, y_train)
print("Test score: %0.3f" % (knn_scaled.score(X_test_scaled, y_test)))
```

```
Train score: 0.798
Test score: 0.686
```

- The scores with scaled data are better compared to the unscaled data in case of k-NNs.
- I am not carrying out cross-validation here for a reason that we'll look into soon.
- Note that I am a bit sloppy here and using the test set several times for teaching purposes. But when you build an ML pipeline, please do assessment on the test set only once.

Common preprocessing techniques

Some commonly performed feature transformation include:

- Imputation: Tackling missing values
- Scaling: Scaling of numeric features
- One-hot encoding: Tackling categorical variables

We could have one lecture on each of them! In this lesson our goal is to getting familiar with them so that we can use them to build ML pipelines.

In the next part of this lecture, we'll build an ML pipeline using [California housing prices regression dataset (https://www.kaggle.com/harrywang/housing)](https://www.kaggle.com/harrywang/housing). In the process, we will talk about different feature transformations and how can we apply them so that we do not violate the golden rule.

Imputation and scaling [video (https://youtu.be/G2IXbVzKlt8)]

Dataset, splitting, and baseline

We'll be working on California housing prices regression dataset (https://www.kaggle.com/harrywang/housing) to demonstrate these feature transformation techniques. The task is to predict median house values in Californian districts, given a number of features from these districts. If you are running the notebook on your own, you'll have to download the data and put it in the data directory.

In [12]:

```
housing_df = pd.read_csv("../data/housing.csv")
train_df, test_df = train_test_split(housing_df, test_size=0.1, random_

train_df.head()
```

Out[12]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
6051	-117.75	34.04	22.0	2948.0	636.0	2600.0	602.0
20113	-119.57	37.94	17.0	346.0	130.0	51.0	20.0
14289	-117.13	32.74	46.0	3355.0	768.0	1457.0	708.0
13665	-117.31	34.02	18.0	1634.0	274.0	899.0	285.0
14471	-117.23	32.88	18.0	5566.0	1465.0	6303.0	1458.0

Some column values are mean/median but some are not.

Let's add some new features to the dataset which could help predicting the target: `median_house_value` .

In [13]:

```
train_df = train_df.assign(
    rooms_per_household=train_df["total_rooms"] / train_df["households"
)
test_df = test_df.assign(
    rooms_per_household=test_df["total_rooms"] / test_df["households"]
)

train_df = train_df.assign(
    bedrooms_per_household=train_df["total_bedrooms"] / train_df["house
)
test_df = test_df.assign(
    bedrooms_per_household=test_df["total_bedrooms"] / test_df["househo
)

train_df = train_df.assign(
    population_per_household=train_df["population"] / train_df["househo
)
test_df = test_df.assign(
    population_per_household=test_df["population"] / test_df["household
)
```

In [14]:

```
train_df.head()
```

Out[14]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
6051	-117.75	34.04	22.0	2948.0	636.0	2600.0	602.0
20113	-119.57	37.94	17.0	346.0	130.0	51.0	20.0
14289	-117.13	32.74	46.0	3355.0	768.0	1457.0	708.0
13665	-117.31	34.02	18.0	1634.0	274.0	899.0	285.0
14471	-117.23	32.88	18.0	5566.0	1465.0	6303.0	1458.0

When is it OK to do things before splitting?

- Here it would have been OK to add new features before splitting because we are not using any global information about the feature values (e.g. the mean of a feature across rows), but only looking at one row at a time.
- But just to be safe and to avoid accidentally breaking the golden rule, it's better to do it after splitting.
- Question: Should we remove `total_rooms` , `total_bedrooms` , and `population` columns?
    - Probably. But I am keeping them in this lecture. You could experiment with removing them and examine whether results change.

Exploratory Data Analysis (EDA)

In [15]:
```
train_df.head()
```

Out[15]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
6051	-117.75	34.04	22.0	2948.0	636.0	2600.0	602.0
20113	-119.57	37.94	17.0	346.0	130.0	51.0	20.0
14289	-117.13	32.74	46.0	3355.0	768.0	1457.0	708.0
13665	-117.31	34.02	18.0	1634.0	274.0	899.0	285.0
14471	-117.23	32.88	18.0	5566.0	1465.0	6303.0	1458.0

The feature scales are quite different.

In [16]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 18576 entries, 6051 to 19966
Data columns (total 13 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   longitude                 18576 non-null  float64
 1   latitude                  18576 non-null  float64
 2   housing_median_age        18576 non-null  float64
 3   total_rooms               18576 non-null  float64
 4   total_bedrooms            18391 non-null  float64
 5   population                18576 non-null  float64
 6   households                18576 non-null  float64
 7   median_income             18576 non-null  float64
 8   median_house_value        18576 non-null  float64
 9   ocean_proximity           18576 non-null  object 
 10  rooms_per_household       18576 non-null  float64
 11  bedrooms_per_household    18391 non-null  float64
 12  population_per_household  18576 non-null  float64
dtypes: float64(12), object(1)
memory usage: 2.0+ MB
```

We have one categorical feature and all other features are numeric features.

In [17]:
```python
train_df.describe()
```

Out[17]:

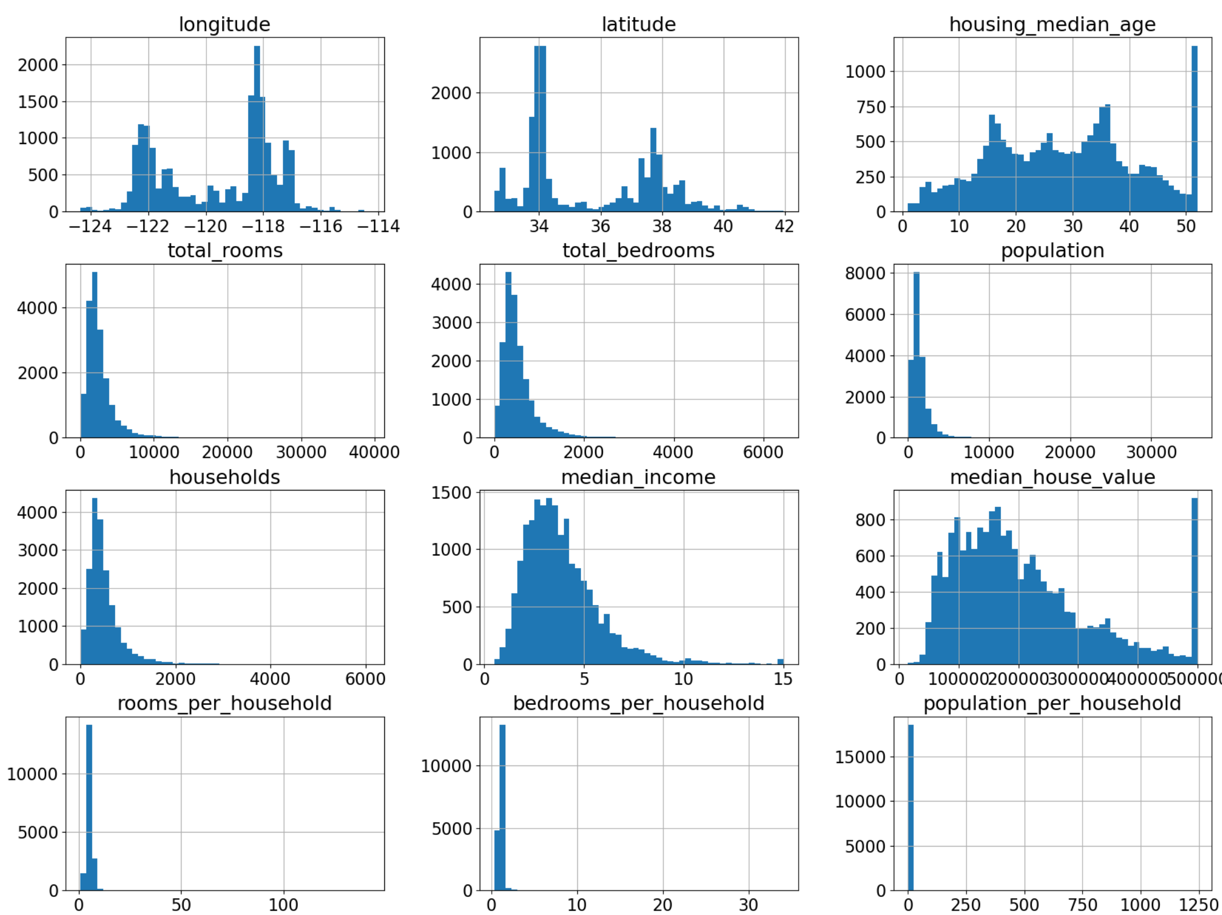
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
count	18576.000000	18576.000000	18576.000000	18576.000000	18391.000000	18576.00000
mean	-119.565888	35.627966	28.622255	2635.749677	538.229786	1428.57816
std	1.999622	2.134658	12.588307	2181.789934	421.805266	1141.66480
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.00000
25%	-121.790000	33.930000	18.000000	1449.000000	296.000000	788.00000
50%	-118.490000	34.250000	29.000000	2127.000000	435.000000	1167.00000
75%	-118.010000	37.710000	37.000000	3145.000000	647.000000	1727.00000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.00000

- Seems like total_bedrooms column has some missing values.
- This must have affected our new feature `bedrooms_per_household` as well.

In [18]:
```python
housing_df["total_bedrooms"].isnull().sum()
```

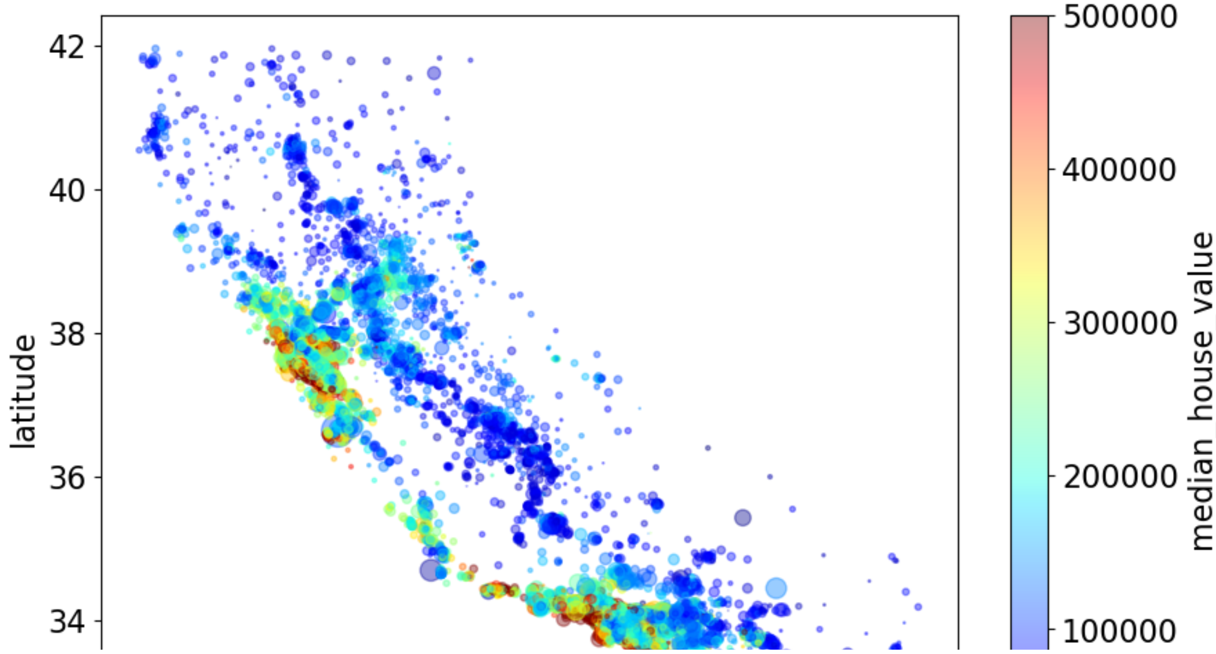
Out[18]: 207

In [19]:
```python
## (optional)
train_df.hist(bins=50, figsize=(20, 15));
```



In [20]:

```
## (optional)
train_df.plot(
    kind="scatter",
    x="longitude",
    y="latitude",
    alpha=0.4,
    s=train_df["population"] / 100,
    figsize=(10, 7),
    c="median_house_value",
    cmap=plt.get_cmap("jet"),
    colorbar=True,
    sharex=False,
);
```



Was that all the transformations we need to apply to the dataset?

Here is what we see from the exploratory data analysis (EDA).

- Some missing values in `total_bedrooms` column.
- Scales are quite different across columns.
- Categorical variable `ocean_proximity` .

Read about [preprocessing techniques implemented in `scikit-learn` (https://scikit-learn.org/stable/modules/preprocessing.html)](https://scikit-learn.org/stable/modules/preprocessing.html).

In [21]:

```
# We are droping the categorical variable ocean_proximity for now. We'l
X_train = train_df.drop(columns=["median_house_value", "ocean_proximity
y_train = train_df["median_house_value"]

X_test = test_df.drop(columns=["median_house_value", "ocean_proximity"]
y_test = test_df["median_house_value"]
```

Let's first run our baseline model `DummyRegressor`

In [22]:
```
results_dict = {}  # dictionary to store our results for different mode
```

In [23]:
```
def mean_std_cross_val_scores(model, X_train, y_train, **kwargs):
    """
    Returns mean and std of cross validation

    Parameters
    ----------
    model :
        scikit-learn model
    X_train : numpy array or pandas DataFrame
        X in the training data
    y_train :
        y in the training data

    Returns
    ----------
        pandas Series with mean scores from cross_validation
    """

    scores = cross_validate(model, X_train, y_train, **kwargs)

    mean_scores = pd.DataFrame(scores).mean()
    std_scores = pd.DataFrame(scores).std()
    out_col = []

    for i in range(len(mean_scores)):
        out_col.append((f"%0.3f (+/- %0.3f)" % (mean_scores[i], std_sco

    return pd.Series(data=out_col, index=mean_scores.index)
```

In [24]:
```
dummy = DummyRegressor(strategy="median")
results_dict["dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True
)
```

In [25]:
```
pd.DataFrame(results_dict)
```

Out[25]:

	dummy
fit_time	0.002 (+/- 0.001)
score_time	0.001 (+/- 0.000)
test_score	-0.055 (+/- 0.012)
train_score	-0.055 (+/- 0.001)

Imputation

In [27]:
```
knn = KNeighborsRegressor()
knn.fit(X_train, y_train)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[27], line 2
      1 knn = KNeighborsRegressor()
----> 2 knn.fit(X_train, y_train)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/neighbors/_regression.py:210, in KNeighborsRegressor.fit(self, X, y)
    191 """Fit the k-nearest neighbors regressor from the training dataset.
    192 
    193 Parameters
   (...)
    206     The fitted k-nearest neighbors regressor.
    207 """
    208 self.weights = _check_weights(self.weights)
--> 210 return self._fit(X, y)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/neighbors/_base.py:407, in NeighborsBase._fit(self, X, y)
    405 if self._get_tags()["requires_y"]:
    406     if not isinstance(X, (KDTree, BallTree, NeighborsBase)):
--> 407         X, y = self._validate_data(
    408             X, y, accept_sparse="csr", multi_output=True, order="C"
    409         )
    411     if is_classifier(self):
    412         # Classification targets require a specific format
    413         if y.ndim == 1 or y.ndim == 2 and y.shape[1] == 1:

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/base.py:596, in BaseEstimator._validate_data(self, X, y, reset, validate_separately, **check_params)
    594         y = check_array(y, input_name="y", **check_y_params)
    595     else:
--> 596         X, y = check_X_y(X, y, **check_params)
    597     out = X, y
    599 if not no_val_X and check_params.get("ensure_2d", True):

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/utils/validation.py:1074, in check_X_y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, multi_output, ensure_min_samples, ensure_min_features, y_numeric, estimator)
   1069         estimator_name = _check_estimator_name(estimator)
   1070     raise ValueError(
   1071         f"{estimator_name} requires y to be passed, but the target y is None"
   1072     )
-> 1074 X = check_array(
   1075     X,
   1076     accept_sparse=accept_sparse,
   1077     accept_large_sparse=accept_large_sparse,
   1078     dtype=dtype,
   1079     order=order,
   1080     copy=copy,
```

```
   1081     force_all_finite=force_all_finite,
   1082     ensure_2d=ensure_2d,
   1083     allow_nd=allow_nd,
   1084     ensure_min_samples=ensure_min_samples,
   1085     ensure_min_features=ensure_min_features,
   1086     estimator=estimator,
   1087     input_name="X",
   1088 )
   1090 y = _check_y(y, multi_output=multi_output, y_numeric=y_numeric, e
stimator=estimator)
   1092 check_consistent_length(X, y)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/util
s/validation.py:899, in check_array(array, accept_sparse, accept_large_sp
arse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_m
in_samples, ensure_min_features, estimator, input_name)
    893         raise ValueError(
    894             "Found array with dim %d. %s expected <= 2."
    895             % (array.ndim, estimator_name)
    896         )
    898     if force_all_finite:
--> 899         _assert_all_finite(
    900             array,
    901             input_name=input_name,
    902             estimator_name=estimator_name,
    903             allow_nan=force_all_finite == "allow-nan",
    904         )
    906 if ensure_min_samples > 0:
    907     n_samples = _num_samples(array)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/util
s/validation.py:146, in _assert_all_finite(X, allow_nan, msg_dtype, estim
ator_name, input_name)
    124         if (
    125             not allow_nan
    126             and estimator_name
   (...)
    130             # Improve the error message on how to handle missing
values in
    131             # scikit-learn.
    132             msg_err += (
    133                 f"\n{estimator_name} does not accept missing valu
es"
    134                 " encoded as NaN natively. For supervised learnin
g, you might want"
   (...)
    144                 "#estimators-that-handle-nan-values"
    145             )
--> 146         raise ValueError(msg_err)
    148 # for object dtype data, we only check for NaNs (GH-13254)
    149 elif X.dtype == np.dtype("object") and not allow_nan:

ValueError: Input X contains NaN.
KNeighborsRegressor does not accept missing values encoded as NaN nativel
y. For supervised learning, you might want to consider sklearn.ensemble.H
istGradientBoostingClassifier and Regressor which accept missing values e
ncoded as NaNs natively. Alternatively, it is possible to preprocess the
```

data, for instance by using an imputer transformer in a pipeline or drop samples with missing values. See https://scikit-learn.org/stable/modules/impute.html (https://scikit-learn.org/stable/modules/impute.html) You can find a list of all estimators that handle NaN values at the following page: https://scikit-learn.org/stable/modules/impute.html#estimators-that-handle-nan-values (https://scikit-learn.org/stable/modules/impute.html#estimators-that-handle-nan-values)

What's the problem?

```
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

- The classifier is not able to deal with missing values (NaNs).
- What are possible ways to deal with the problem?
  - Delete the rows?
  - Replace them with some reasonable values?

- `SimpleImputer` is a transformer in `sklearn` to deal with this problem. For example:
  - You can impute missing values in categorical columns with the most frequent value.
  - You can impute the missing values in numeric columns with the mean or median of the column.

In [28]:
```
X_train.sort_values("bedrooms_per_household")
```

Out[28]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
20248	-119.23	34.25	28.0	26.0	3.0	29.0	9.(
12649	-121.47	38.51	52.0	20.0	4.0	74.0	9.(
3125	-117.76	35.22	4.0	18.0	3.0	8.0	6.(
12138	-117.22	33.87	16.0	56.0	7.0	39.0	14.(
8219	-118.21	33.79	33.0	32.0	18.0	96.0	36.(
...	...	...	...	...	...	...	..
4591	-118.28	34.06	42.0	2472.0	NaN	3795.0	1179.(
19485	-120.98	37.66	10.0	934.0	NaN	401.0	255.(
6962	-118.05	33.99	38.0	1619.0	NaN	886.0	357.(
14970	-117.01	32.74	31.0	3473.0	NaN	2098.0	677.(
7763	-118.10	33.91	36.0	726.0	NaN	490.0	130.(

18576 rows × 11 columns

In [29]:
```
X_train.shape
X_test.shape
```

Out[29]:
```
(2064, 11)
```

In [32]:
```
imputer = SimpleImputer(strategy="median")
imputer.fit(X_train)
X_train_imp = imputer.transform(X_train)
X_test_imp = imputer.transform(X_test)
```

- Let's check whether the NaN values have been replaced or not
- Note that `imputer.transform` returns an `numpy` array and not a dataframe

In [37]:
```
df = pd.DataFrame(X_train_imp, columns=X_train.columns, index=X_train.i
df.loc[7763]
# df.loc[7763]
df.isnull().sum()
```

Out[37]:
```
longitude                   0
latitude                    0
housing_median_age          0
total_rooms                 0
total_bedrooms              0
population                  0
households                  0
median_income               0
rooms_per_household         0
bedrooms_per_household      0
population_per_household    0
dtype: int64
```

- Now the k-NN runs!
- The training error is bad though...

In [38]:
```
knn = KNeighborsRegressor()
knn.fit(X_train_imp, y_train)
knn.score(X_train_imp, y_train)
```

Out[38]:
```
0.5085407150708963
```

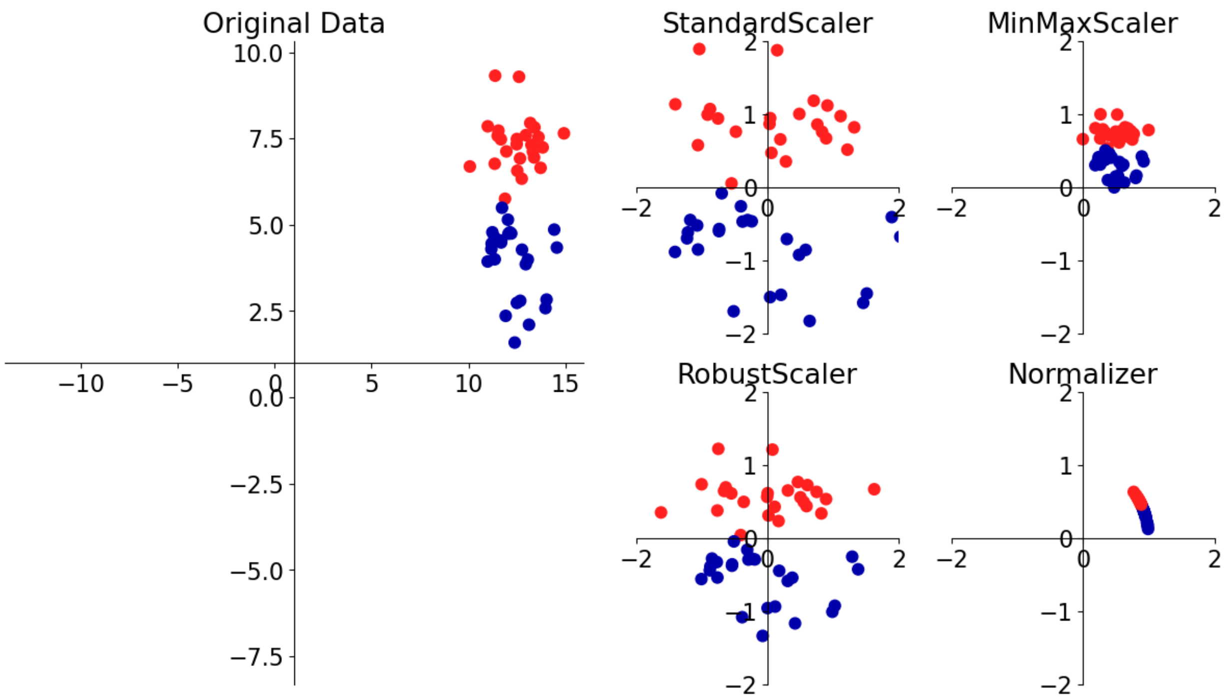
Scaling

- This problem affects a large number of ML methods.
- A number of approaches to this problem. We are going to look into the two most popular ones.

Approach	What it does	How to update X (but see below!)	
normalization	sets range to $[0, 1]$	`X -= np.min(X,axis=0)` `X /= np.max(X,axis=0)`	`MinMaxScaler()` (https://scikit-learn.org/stable/modul

In [39]:

```
# [source](https://amueller.github.io/COMS4995-s19/slides/aml-05-prepro
mglearn.plots.plot_scaling()
```



In [40]:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
```

In [41]:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_imp)  # fit once
X_test_scaled = scaler.transform(X_test_imp)        # Transform test se
pd.DataFrame(X_train_scaled, columns=X_train.columns)
```

Out[41]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	househo
0	0.908140	-0.743917	-0.526078	0.143120	0.235339	1.026092	0.2661
1	-0.002057	1.083123	-0.923283	-1.049510	-0.969959	-1.206672	-1.2533
2	1.218207	-1.352930	1.380504	0.329670	0.549764	0.024896	0.5428
3	1.128188	-0.753286	-0.843842	-0.459154	-0.626949	-0.463877	-0.5614
4	1.168196	-1.287344	-0.843842	1.343085	2.210026	4.269688	2.5009
...	...	...	...	...	...	...	
18571	0.733102	-0.804818	0.586095	-0.875337	-0.243446	-0.822136	-0.9661
18572	1.163195	-1.057793	-1.161606	0.940194	0.609314	0.882438	0.7282
18573	-1.097293	0.797355	-1.876574	0.695434	0.433046	0.881563	0.5141
18574	-1.437367	1.008167	1.221622	-0.499947	-0.484029	-0.759944	-0.4544
18575	0.242996	0.272667	-0.684960	-0.332190	-0.353018	-0.164307	-0.3969

18576 rows × 11 columns

In [42]:

```
knn = KNeighborsRegressor()
knn.fit(X_train_scaled, y_train)
knn.score(X_train_scaled, y_train)
```

Out[42]: 0.8090877831586284

- Big difference in the KNN training performance after scaling the data.
- But we saw last week that training score doesn't tell us much. We should look at the cross-validation score.

? ? Questions for class discussion

True/False on scaling and imputation

1. `StandardScaler` ensures a fixed range (i.e., minimum and maximum values) for the features.
2. `StandardScaler` calculates mean and standard deviation for each feature separately.
3. In general, it's a good idea to apply scaling on numeric features before training k-NN or SVM RBF models.
4. After normalization, all numerical features in training and test set will have values between 0 and 1.
5. It is OK to modify features before splitting into training and testing set if the modification is row-wise.

```
1. False
2. True
3. True
4. False (not in the test set)
5. True (but we don't recommend it)
```

More True/False on scaling and imputation

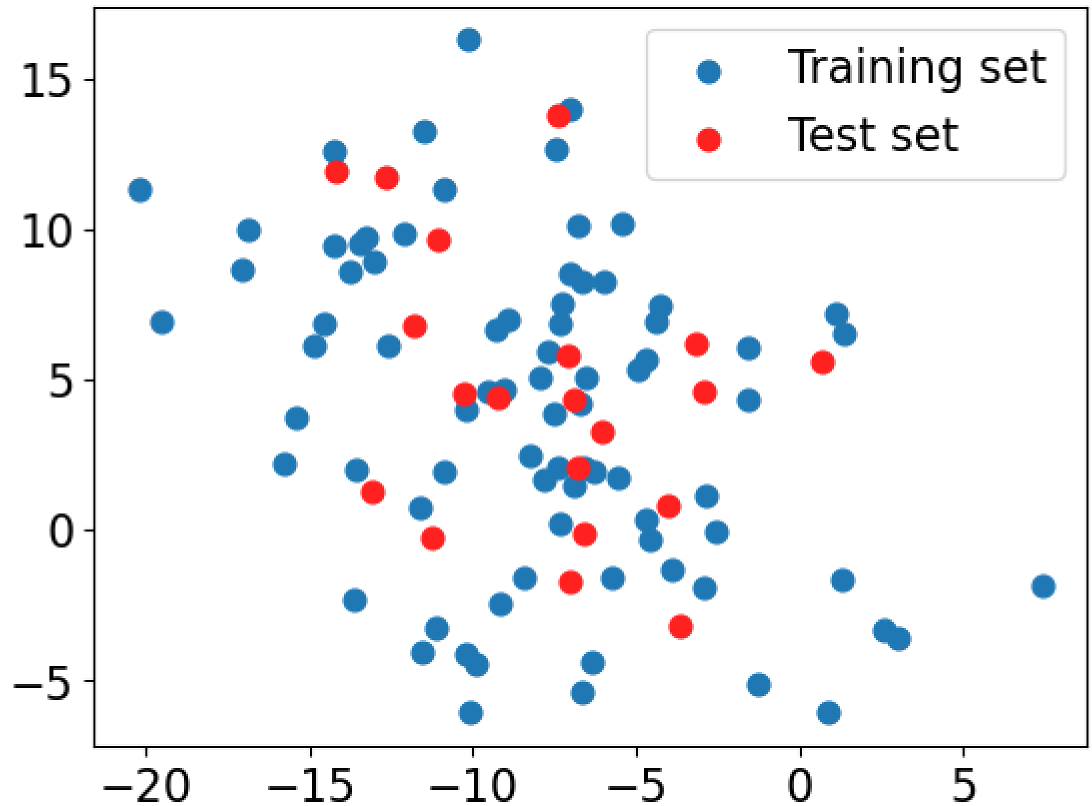
4. The transformers such as `StandardScaler` or `SimpleImputer` in `scikit-learn` return a dataframe with transformed features.
5. The transformed feature values might be hard to interpret for humans.
6. After applying `SimpleImputer` , the transformed data has a different shape than the original data.

Consider a toy data with the following two columns. If you apply `StandardScaler` on this data, both columns A and B will end up being identical. True or False?

In [ ]:
```
ex_test_cols = np.array([[10, -2], [20, -1], [30, 0], [40, 1], [50, 2]]
ex_test_df = pd.DataFrame(data=ex_test_cols, columns=["A", "B"])
ex_test_df
```

Let's create some synthetic data.

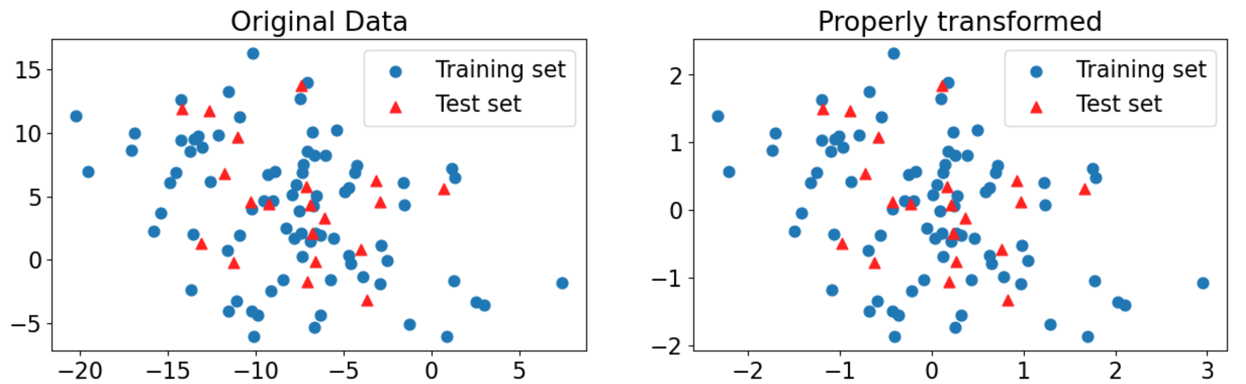
In [43]:
```
from sklearn.datasets import make_blobs, make_classification

# make synthetic data
X, y = make_blobs(n_samples=100, centers=3, random_state=12, cluster_st
# split it into training and test sets
X_train_toy, X_test_toy, y_train_toy, y_test_toy = train_test_split(
    X, y, random_state=5, test_size=0.2
)
plt.scatter(X_train_toy[:, 0], X_train_toy[:, 1], label="Training set",
plt.scatter(
    X_test_toy[:, 0], X_test_toy[:, 1], color=mglearn.cm2(1), label="Te
)
plt.legend(loc="upper right");
```



Let's transform the data using `StandardScaler` and examine how the data looks like.

In [44]:
```
scaler = StandardScaler()
train_transformed = scaler.fit_transform(X_train_toy)
test_transformed = scaler.transform(X_test_toy)
```

In [45]:
```
plot_original_scaled(X_train_toy, X_test_toy, train_transformed, test_t
```



In [46]:
```
knn = KNeighborsClassifier()
knn.fit(train_transformed, y_train_toy)
print(f"Training score: {knn.score(train_transformed, y_train_toy):.2f}
print(f"Test score: {knn.score(test_transformed, y_test_toy):.2f}")
```

```
Training score: 0.75
Test score: 0.55
```

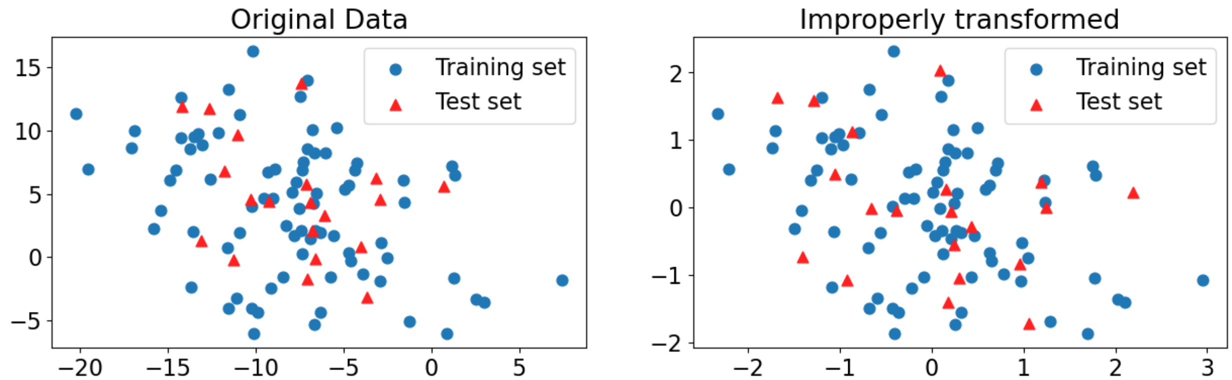
Bad methodology 1: Scaling the data separately (for class discussion)

In [48]:
```
# DO NOT DO THIS! For illustration purposes only.
scaler = StandardScaler()
scaler.fit(X_train_toy)
train_scaled = scaler.transform(X_train_toy)

scaler = StandardScaler()  # Creating a separate object for scaling tes
scaler.fit(X_test_toy)  # Calling fit on the test data
test_scaled = scaler.transform(
    X_test_toy
)  # Transforming the test data using the scaler fit on test data

knn = KNeighborsClassifier()
knn.fit(train_scaled, y_train_toy)
print(f"Training score: {knn.score(train_scaled, y_train_toy):.2f}")
print(f"Test score: {knn.score(test_scaled, y_test_toy):.2f}")
```

```
Training score: 0.75
Test score: 0.50
```

- Is anything wrong in methodology 1? If yes, what is it?

In [49]:
```
plot_original_scaled(
    X_train_toy,
    X_test_toy,
    train_scaled,
    test_scaled,
    title_transformed="Improperly transformed",
)
```



Bad methodology 2: Scaling the data together (for class discussion)

In [51]:
```
X_train_toy.shape, X_test_toy.shape
```

Out[51]:
```
((80, 2), (20, 2))
```

In [52]:
```
# join the train and test sets back together
XX = np.vstack((X_train_toy, X_test_toy))
XX.shape
```

Out[52]:
```
(100, 2)
```

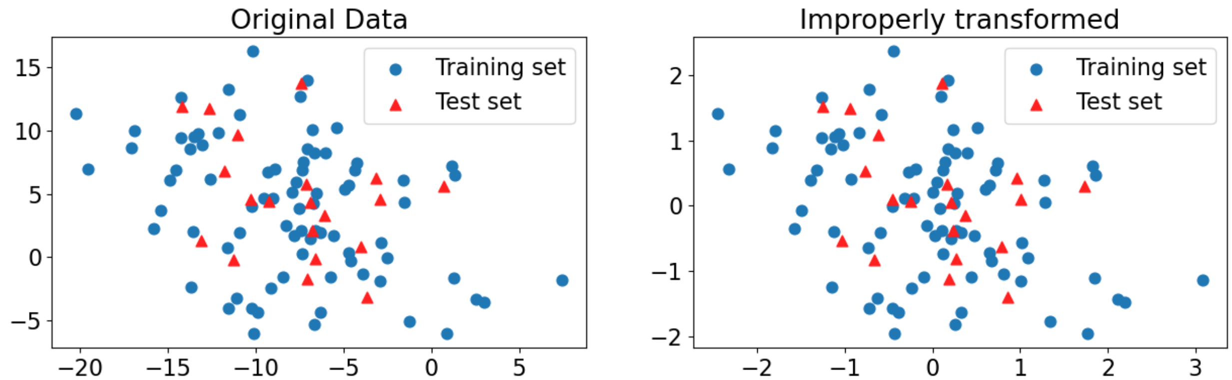
In [53]:
```
scaler = StandardScaler()
scaler.fit(XX)
XX_scaled = scaler.transform(XX)
XX_train = XX_scaled[:80]
XX_test = XX_scaled[80:]
```

In [54]:
```
knn = KNeighborsClassifier()
knn.fit(XX_train, y_train_toy)
print(f"Training score: {knn.score(XX_train, y_train_toy):.2f}")  # Mis
print(f"Test score: {knn.score(XX_test, y_test_toy):.2f}")  # Misleadin
```

```
Training score: 0.75
Test score: 0.55
```

- Is anything wrong in methodology 2? If yes, what is it?

In [55]:
```
plot_original_scaled(
    X_train_toy,
    X_test_toy,
    XX_train,
    XX_test,
    title_transformed="Improperly transformed",
)
```



Not a big difference in the transformed data but if the test set is large it might influence the mean and standard deviation significantly. **Importantly, this breaks the golden rule!**

Methodology 3: Cross validation with already preprocessed data (for class discussion)

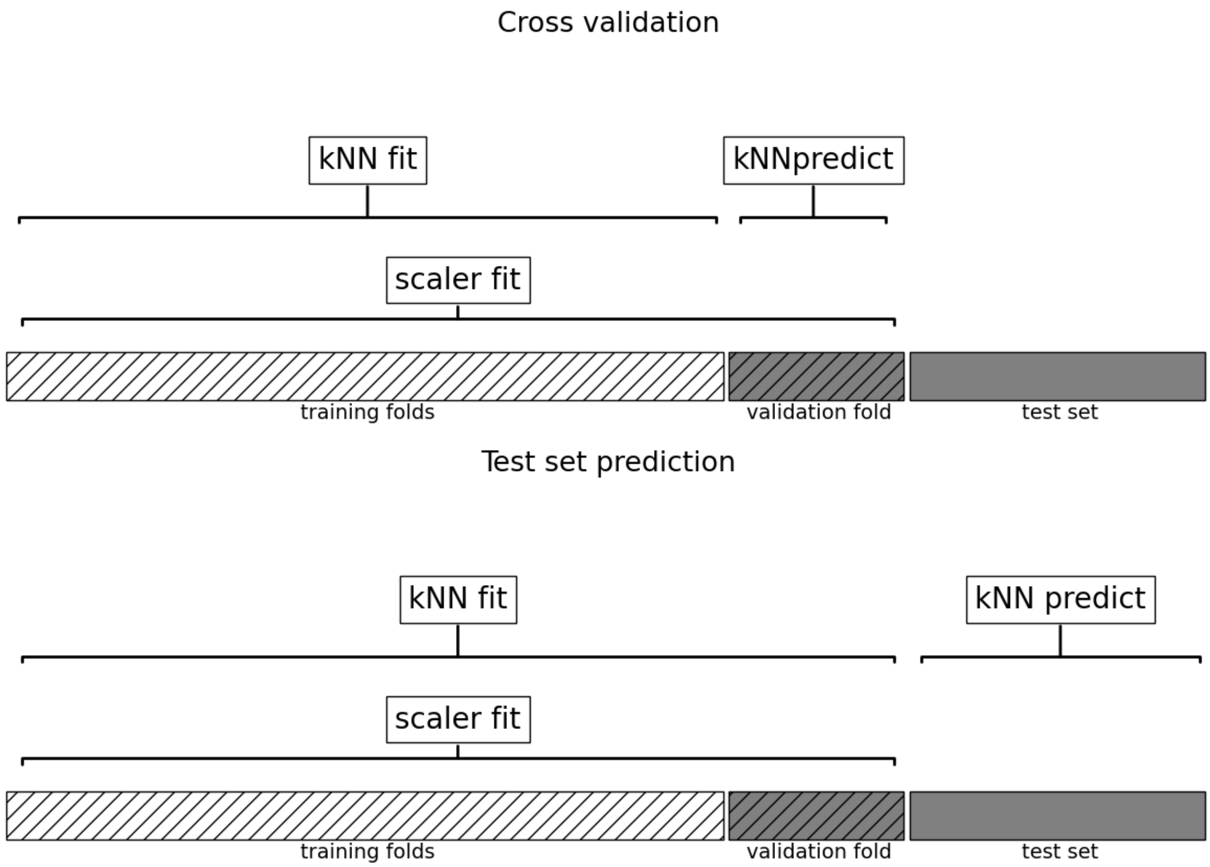
In [56]:
```
knn = KNeighborsClassifier()

scaler = StandardScaler()
scaler.fit(X_train_toy)
X_train_scaled = scaler.transform(X_train_toy)
X_test_scaled = scaler.transform(X_test_toy)
scores = cross_validate(knn, X_train_scaled, y_train_toy, return_train_
pd.DataFrame(scores)
```

Out[56]:

	fit_time	score_time	test_score	train_score
0	0.000557	0.001731	0.6875	0.671875
1	0.000536	0.002388	0.7500	0.671875
2	0.000427	0.001202	0.6875	0.734375
3	0.000385	0.001151	0.6250	0.750000
4	0.000360	0.001179	0.5000	0.687500

- Is there anything wrong in methodology 3? Are we breaking the golden rule here?

In [57]:
```
plot_improper_processing("kNN")
```



In [ ]:
```
plot_proper_processing("kNN")
```

Feature transformations and the golden rule [video ()]

How to carry out cross-validation?

- Last week we saw that cross validation is a better way to get a realistic assessment of the model.
- Let's try cross-validation with transformed data.

In [58]:
```
knn = KNeighborsRegressor()

scaler = StandardScaler()
scaler.fit(X_train_imp)
X_train_scaled = scaler.transform(X_train_imp)
X_test_scaled = scaler.transform(X_test_imp)
scores = cross_validate(knn, X_train_scaled, y_train, return_train_scor
pd.DataFrame(scores)
```

Out[58]:

	fit_time	score_time	test_score	train_score
0	0.012000	0.262788	0.710905	0.803734
1	0.008120	0.227898	0.706893	0.803212
2	0.008050	0.243732	0.711039	0.803030
3	0.008205	0.246907	0.695769	0.806275
4	0.007997	0.181020	0.697941	0.805146

- Do you see any problem here?
- Are we applying `fit_transform` on train portion and `transform` on validation portion in each fold?
  - Here you might be allowing information from the validation set to **leak** into the training step.

- You need to apply the **SAME** preprocessing steps to train/validation.
- With many different transformations and cross validation the code gets unwieldy very quickly.
- Likely to make mistakes and "leak" information.

- In these examples our test accuracies look fine, but our methodology is flawed.
- Implications can be significant in practice!

Pipelines

Can we do this in a more elegant and organized way?

- YES!! Using [`scikit-learn Pipeline` (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html).
- [`scikit-learn Pipeline` (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) allows you to define a "pipeline" of transformers with a final estimator.

Let's combine the preprocessing and model with pipeline

In [59]:
```python
### Simple example of a pipeline
from sklearn.pipeline import Pipeline

pipe = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        ("regressor", KNeighborsRegressor()),
    ]
)
```

- Syntax: pass in a list of steps.
- The last step should be a **model/classifier/regressor**.
- All the earlier steps should be **transformers**.

Alternative and more compact syntax: `make_pipeline`

- Shorthand for `Pipeline` constructor
- Does not permit naming steps
- Instead the names of steps are set to lowercase of their types automatically; `StandardScaler()` would be named as `standardscaler`

In [60]:
```python
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(
    SimpleImputer(strategy="median"), StandardScaler(), KNeighborsRegre
)
```

In [61]:
```python
pipe.fit(X_train, y_train)
```

Out[61]:
```
Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),
                ('standardscaler', StandardScaler()),
                ('kneighborsregressor', KNeighborsRegressor())])
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

- Note that we are passing `X_train` and **not** the imputed or scaled data here.

When you call `fit` on the pipeline, it carries out the following steps:

- Fit `SimpleImputer` on `X_train`
- Transform `X_train` using the fit `SimpleImputer` to create `X_train_imp`
- Fit `StandardScaler` on `X_train_imp`
- Transform `X_train_imp` using the fit `StandardScaler` to create `X_train_imp_scaled`

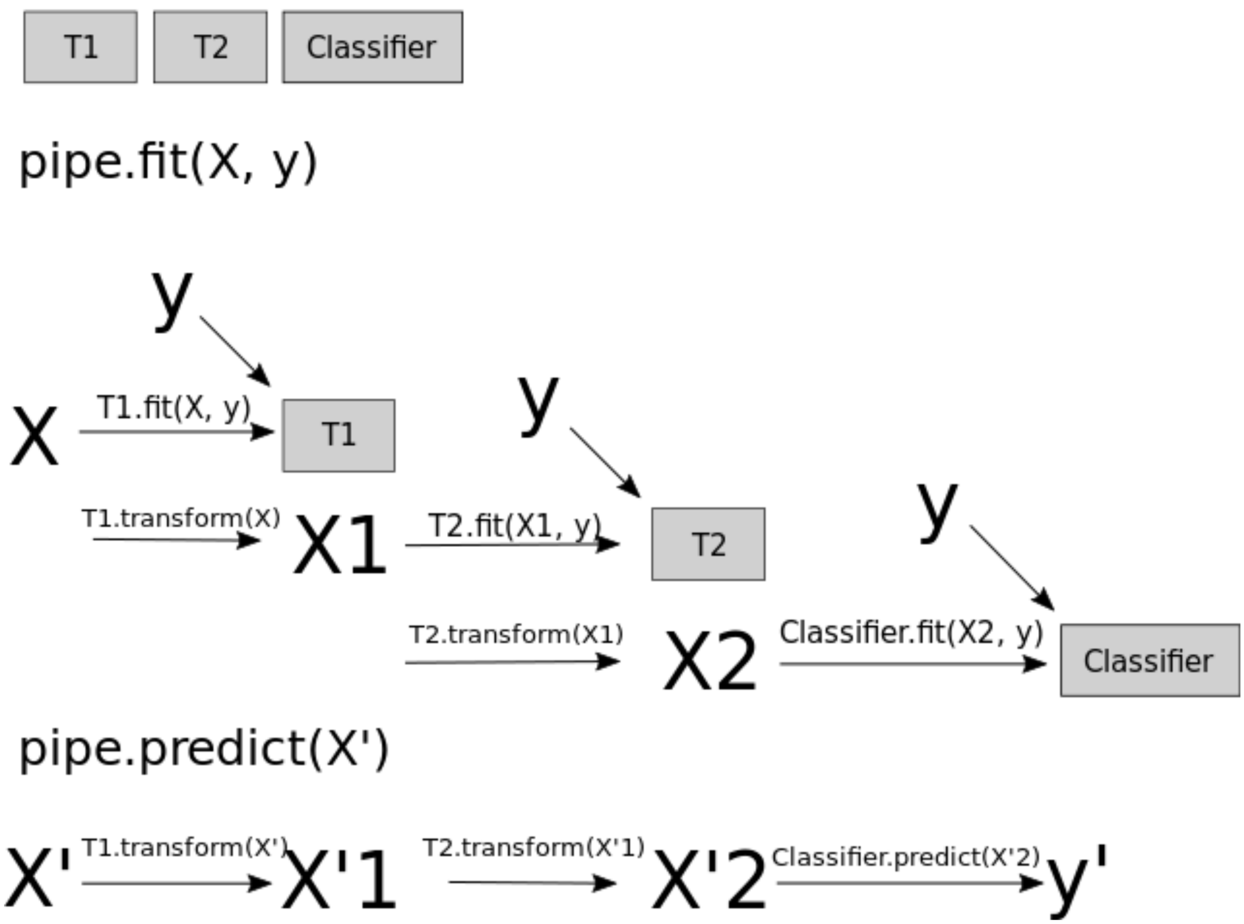
- Fit the model (`KNeighborsRegressor` in our case) on `X_train_imp_scaled`

In [62]:
```
pipe.predict(X_train)
```

Out[62]:
```
array([122460., 115220., 216940., ..., 240420., 254500.,  60420.])
```

Note that we are passing original data to `predict` as well. This time the pipeline is carrying out following steps:

- Transform `X_train` using the fit `SimpleImputer` to create `X_train_imp`
- Transform `X_train_imp` using the fit `StandardScaler` to create `X_train_imp_scaled`
- Predict using the fit model (`KNeighborsRegressor` in our case) on `X_train_imp_scaled` .



[Source (https://amueller.github.io/COMS4995-s20/slides/aml-04-preprocessing/#18)](https://amueller.github.io/COMS4995-s20/slides/aml-04-preprocessing/#18)

Let's try cross-validation with our pipeline

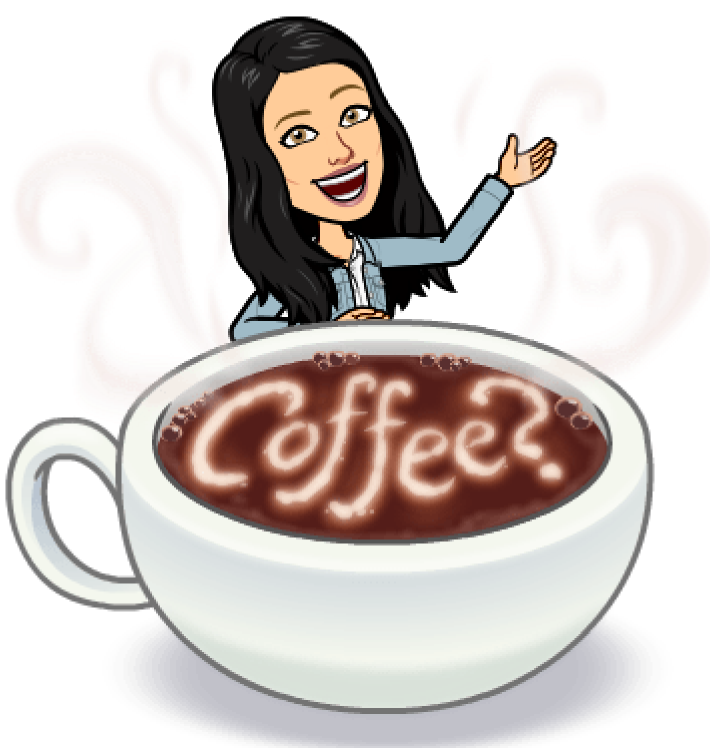
In [63]:
```
results_dict["imp + scaling + knn"] = mean_std_cross_val_scores(
    pipe, X_train, y_train, return_train_score=True
)
pd.DataFrame(results_dict).T
```

Out[63]:

	fit_time	score_time	test_score	train_score
dummy	0.002 (+/- 0.001)	0.001 (+/- 0.000)	-0.055 (+/- 0.012)	-0.055 (+/- 0.001)
imp + scaling + knn	0.028 (+/- 0.003)	0.254 (+/- 0.032)	0.706 (+/- 0.006)	0.806 (+/- 0.005)

Using a `Pipeline` takes care of applying the `fit_transform` on the train portion and only `transform` on the validation portion in each fold.

Break (5 min)



Categorical features [video (https://youtu.be/2mJ9rAhMMI0)]

- Recall that we had dropped the categorical feature `ocean_proximity` feature from the dataframe. But it could potentially be a useful feature in this task.
- Let's create our `X_train` and and `X_test` again by keeping the feature in the data.

In [64]:
```
test_df.head()
```

Out[64]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
19121	-122.64	38.24	40.0	1974.0	410.0	1039.0	398.0
20019	-119.05	36.09	9.0	3297.0	568.0	1749.0	568.0
15104	-116.98	32.85	12.0	3570.0	713.0	3321.0	666.0
3720	-118.42	34.20	27.0	3201.0	970.0	3403.0	948.0
8938	-118.47	34.01	41.0	2704.0	557.0	1047.0	478.0

In [66]:
```
X_train = train_df.drop(columns=["median_house_value"])
y_train = train_df["median_house_value"]

X_test = test_df.drop(columns=["median_house_value"])
y_test = test_df["median_house_value"]
```

- Let's try to build a `KNeighborRegressor` on this data using our pipeline

In [67]:

```
pipe.fit(X_train, X_train)
```

```
---------------------------------------------------------------------------
--
ValueError                                Traceback (most recent call las
t)
Cell In[67], line 1
----> 1 pipe.fit(X_train, X_train)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/pipel
ine.py:378, in Pipeline.fit(self, X, y, **fit_params)
    352 """Fit the model.
    353
    354 Fit all the transformers one after the other and transform the
   (...)
    375     Pipeline with fitted steps.
    376 """
    377 fit_params_steps = self._check_fit_params(**fit_params)
--> 378 Xt = self._fit(X, y, **fit_params_steps)
    379 with _print_elapsed_time("Pipeline", self._log_message(len(self.s
teps) - 1)):
    380     if self._final_estimator != "passthrough":

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/pipel
ine.py:336, in Pipeline._fit(self, X, y, **fit_params_steps)
    334     cloned_transformer = clone(transformer)
    335 # Fit or load from cache the current transformer
--> 336 X, fitted_transformer = fit_transform_one_cached(
    337     cloned_transformer,
    338     X,
    339     y,
    340     None,
    341     message_clsname="Pipeline",
    342     message=self._log_message(step_idx),
    343     **fit_params_steps[name],
    344 )
    345 # Replace the transformer of the step with the fitted
    346 # transformer. This is necessary when loading the transformer
    347 # from the cache.
    348 self.steps[step_idx] = (name, fitted_transformer)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/joblib/memor
y.py:349, in NotMemorizedFunc.__call__(self, *args, **kwargs)
    348 def __call__(self, *args, **kwargs):
--> 349     return self.func(*args, **kwargs)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/pipel
ine.py:870, in _fit_transform_one(transformer, X, y, weight, message_clsn
ame, message, **fit_params)
    868 with _print_elapsed_time(message_clsname, message):
    869     if hasattr(transformer, "fit_transform"):
--> 870         res = transformer.fit_transform(X, y, **fit_params)
    871     else:
    872         res = transformer.fit(X, y, **fit_params).transform(X)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/base.
py:870, in TransformerMixin.fit_transform(self, X, y, **fit_params)
    867     return self.fit(X, **fit_params).transform(X)
    868 else:
```

```
    869     # fit method of arity 2 (supervised transformation)
--> 870     return self.fit(X, y, **fit_params).transform(X)

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/imput
e/_base.py:364, in SimpleImputer.fit(self, X, y)
    355 if self.verbose != "deprecated":
    356     warnings.warn(
    357         "The 'verbose' parameter was deprecated in version "
    358         "1.1 and will be removed in 1.3. A warning will "
   (...)
    361         FutureWarning,
    362     )
--> 364 X = self._validate_input(X, in_fit=True)
    366 # default fill_value is 0 for numerical input and "missing_value"
    367 # otherwise
    368 if self.fill_value is None:

File ~/miniconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/imput
e/_base.py:317, in SimpleImputer._validate_input(self, X, in_fit)
    311 if "could not convert" in str(ve):
    312     new_ve = ValueError(
    313         "Cannot use {} strategy with non-numeric data:\n{}".forma
t(
    314             self.strategy, ve
    315         )
    316     )
--> 317     raise new_ve from None
    318 else:
    319     raise ve

ValueError: Cannot use median strategy with non-numeric data:
could not convert string to float: 'INLAND'
```

- This failed because we have non-numeric data.
- Imagine how k-NN would calculate distances when you have non-numeric features.

Can we use this feature in the model?

- In `scikit-learn` , most algorithms require numeric inputs.
- Decision trees could theoretically work with categorical features.
  - However, the sklearn implementation does not support this.

What are the options?

- Drop the column (not recommended)
  - If you know that the column is not relevant to the target in any way you may drop it.
- We can transform categorical features to numeric ones so that we can use them in the model.
  - [Ordinal encoding (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html) (occasionally recommended)

- One-hot encoding (recommended in most cases) (this lecture)

In [68]:

```python
X_toy = pd.DataFrame(
    {
        "language": [
            "English",
            "Vietnamese",
            "English",
            "Mandarin",
            "English",
            "English",
            "Mandarin",
            "English",
            "Vietnamese",
            "Mandarin",
            "French",
            "Spanish",
            "Mandarin",
            "Hindi",
        ]
    }
)
X_toy
```

Out[68]:

	language
0	English
1	Vietnamese
2	English
3	Mandarin
4	English
5	English
6	Mandarin
7	English
8	Vietnamese
9	Mandarin
10	French
11	Spanish
12	Mandarin
13	Hindi

Ordinal encoding (occasionally recommended)

- Here we simply assign an integer to each of our unique categorical labels.
- We can use sklearn's `OrdinalEncoder` [(https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html).

In [69]:

```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder()
enc.fit(X_toy)
X_toy_ord = enc.transform(X_toy)
df = pd.DataFrame(
    data=X_toy_ord,
    columns=["language_enc"],
    index=X_toy.index,
)
pd.concat([X_toy, df], axis=1)
```

Out[69]:

	language	language_enc
0	English	0.0
1	Vietnamese	5.0
2	English	0.0
3	Mandarin	3.0
4	English	0.0
5	English	0.0
6	Mandarin	3.0
7	English	0.0
8	Vietnamese	5.0
9	Mandarin	3.0
10	French	1.0
11	Spanish	4.0
12	Mandarin	3.0
13	Hindi	2.0

What's the problem with this approach?

- We have imposed ordinality on the categorical data.
- For example, imagine when you are calculating distances. Is it fair to say that French and Hindi are closer than French and Spanish?
- In general, label encoding is useful if there is ordinality in your data and capturing it is important for your problem, e.g., `[cold, warm, hot]`.

One-hot encoding (OHE)

- Create new binary columns to represent our categories.
- If we have c categories in our column.
  - We create c new binary columns to represent those categories.
- Example: Imagine a language column which has the information on whether you
- We can use sklearn's [`OneHotEncoder` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) to do so.

One-hot encoding is called one-hot because only one of the newly created features is 1 for each data point.

In [70]:
```python
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(handle_unknown="ignore", sparse=False)
enc.fit(X_toy)
X_toy_ohe = enc.transform(X_toy)
pd.DataFrame(
    data=X_toy_ohe,
    columns=enc.get_feature_names_out(["language"]),
    index=X_toy.index,
)
```

Out[70]:

	language_English	language_French	language_Hindi	language_Mandarin	language_Spanish	langu
0	1.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	
2	1.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	1.0	0.0	
4	1.0	0.0	0.0	0.0	0.0	
5	1.0	0.0	0.0	0.0	0.0	
6	0.0	0.0	0.0	1.0	0.0	
7	1.0	0.0	0.0	0.0	0.0	
8	0.0	0.0	0.0	0.0	0.0	
9	0.0	0.0	0.0	1.0	0.0	
10	0.0	1.0	0.0	0.0	0.0	
11	0.0	0.0	0.0	0.0	1.0	
12	0.0	0.0	0.0	1.0	0.0	
13	0.0	0.0	1.0	0.0	0.0	

Let's do it on our housing data

In [71]:
```python
ohe = OneHotEncoder(sparse=False, dtype="int")
ohe.fit(X_train[["ocean_proximity"]])
X_imp_ohe_train = ohe.transform(X_train[["ocean_proximity"]])
```

- We can look at the new features created using `categories_` attribute

In [72]:
```
ohe.categories_
```

Out[72]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

In [73]:
```
transformed_ohe = pd.DataFrame(
    data=X_imp_ohe_train,
    columns=ohe.get_feature_names_out(["ocean_proximity"]),
    index=X_train.index,
)
transformed_ohe
```

Out[73]:

	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEA BA
6051	0	1	0	
20113	0	1	0	
14289	0	0	0	
13665	0	1	0	
14471	0	0	0	
...	...	...	...	
7763	1	0	0	
15377	1	0	0	
17730	1	0	0	
15725	0	0	0	
19966	0	1	0	

18576 rows × 5 columns

One-hot encoded variables are also referred to as **dummy variables**. You will often see people using `get_dummies` [method of pandas (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html)](https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html) to convert categorical variables into dummy variables. That said, using `sklearn`'s `OneHotEncoder` has the advantage of making it easy to treat training and test set in a consistent way.

? ? Questions for class discussion

True/False: Pipelines and one-hot encoding

1. You can "glue" together imputation and scaling of numeric features and `scikit-learn` classifier object within a single pipeline.

2. You can "glue" together scaling of numeric features, one-hot encoding of categorical features, and `scikit-learn` classifier object within a single pipeline.
3. Pipelines will `fit` and `transform` on the training fold and only `transform` on the validation fold during cross-validation.
4. What's the better encoding for weather labels such as "sunny", "overcast" and "rainy"?

```
1. True
2. Not yet (we'll find a way)
3. True
4. One-hot
```

What did we learn today?

- Motivation for preprocessing
- Common preprocessing steps
  - Imputation
  - Scaling
  - One-hot encoding
- Golden rule in the context of preprocessing
- Building simple supervised machine learning pipelines using `sklearn.pipeline.make_pipeline`.

Problem: Different transformations on different columns

- How do we put this together with other columns in the data before fitting the regressor?
- Before we fit our regressor, we want to apply different transformations on different columns
  - Numeric columns
    - imputation
    - scaling
  - Categorical columns
    - imputation
    - one-hot encoding

Coming up: sklearn's [`ColumnTransformer`](https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html) (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html)!!

In [ ]:
```
1
```